Grand Unified Socket Interface

User's Manual

Version 1.55
Last Updated:20Apr95

Matthias Neeracher <neeri@iis.ee.ethz.ch>

Introduction

GUSI is an extension and partial replacement of the MPW runtime library. Its main objective is to provide a more or less simple and consistent interface across the following *communication domains*:

Files

Ordinary Macintosh files and MPW pseudo devices.

Unix

Memory based communication within a single machine (This name exists for historical reasons).

Appletalk

ADSP (and possibly in the future DDP) communication over a network.

PPC

Local and remote connections with the System 7 PPC Toolbox

Internet

TCP and UDP connections over MacTCP.

PAP

Connections with the Printer Access Protocol, typically to a networked PostScript printer.

Additionally, GUSI adds some UNIX library calls dealing with files which were missing, like *chdir()*, *getcwd()*, *symlink()*, and *readlink()*, and changes a few other library calls to behave more like their UNIX counterparts.

The most recent version of GUSI may be obtained by anonymous ftp from ftp.switch.ch in the directory software/mac/src/mpw_c.

There is also a mailing list devoted to discussions about GUSI. You can join the list by sending email to <*gusi-request@iis.ee.ethz.ch*>.

User's Manual

For ease of access, the manual has been split up into a number of sections:

GUSI_Install Installing and using the GUSI headers and libraries

GUSI_Common Routines common to all GUSI socket families

GUSI_Files Routines specific to the file system

GUSI_Unix Routines specific to memory based (UNIX) sockets

GUSI_Appletalk Routines specific to AppleTalk sockets

GUSI_PPC Routines specific to PPC Toolbox sockets

GUSI_INET Routines specific to internet sockets

GUSI_PAP Routines specific to PAP sockets

GUSI_Misc Miscellaneous routines

GUSI_Advanced Advanced techniques

Copying

Copyright (C) 1992-1995 Matthias Neeracher

Permission is granted to anyone to use this software for any purpose on any computer system, and to redistribute it freely, subject to the following restrictions:

The author is not responsible for the consequences of use of this software, no matter how awful, even if they arise from defects in it.

The origin of this software must not be misrepresented, either by explicit claim or by omission.

Altered versions must be plainly marked as such, and must not be misrepresented as being the original software.

Design Objectives

GUSI was developed according to at least three mutually conflicting standards:

The definition of the existing C library.

The behavior of the corresponding UNIX calls. While my original guideline was a set of discarded SunOS manuals, my current reference point is the ANSI/IEEE POSIX standard (A borrowed copy of the 1988 edition, if you really want to know; feel free to donate me a copy of the 1992 edition). The behaviour of the socket calls is, of course, modeled after their BSD implementation.

The author's judgement, prejudices, laziness, and limited resources.

In general, the behavior of the corresponding POSIX/BSD library call was implemented, since this faciliates porting UNIXish utilities to the Macintosh.

Acknowledgements

I would like to thank all who have agreed to beta test this code and who have provided feedback.

The TCP/IP code in GUSIINET.cp, GUSITCP.cp, and GUSIUDP.cp is derived from a socket library written by Charlie Reiman *<reiman@talisman.kaleida.com>*, which in turn is based on code written by Tom Milligan *<milligan@madhaus.utcs.utoronto.ca>*.

The PAP code in GUSIPAP.cp is derived from code written by Sak Wathanasin *<sw@nan.co.uk>*.

Martin Heller *<heller@gis.geogr.unizh.ch>* suggested to move the documentation to HTML and wrote the HTML to RTF converter. Ed Draper *<draper@usis.com>* provided the PDF translation.

Many of the header files in the :include: subdirectory are borrowed from BSD 4.4-lite, therefore:This product includes software developed by the University of California, Berkeley and its contributors.

Installing and using GUSI

This section discusses how you can install `GUSI` on your disk and use it for your programs.

To install `GUSI`, change in the MPW Shell to its directory and type:

```
BuildProgram Install <Enter>
```

This will install all necessary files in `{CIncludes}`, `{CLibraries}`, and `{RIncludes}`, respectively. It will also install `/etc/services` in your preferences folder, prompting you if you have an older version there.

This requires that you have MPW Perl installed, which is available in the same ftp directory as `GUSI`.

To use `GUSI`, include one or more of the following header files in your program:

GUSI.h

The main file. This includes almost everything else.

TFileSpec.h

`FSSpec` manipulation routines.

dirent.h

Routines to access all entries in a directory.

netdb.h

Looking up TCP/IP host names.

netinet/in.h

The address format for TCP/IP sockets.

sys/errno.h

The errors codes returned by GUSI routines.

sys/ioctl.h

Codes to pass to *`ioctl()`*.

sys/socket.h

Data types for socket calls.

sys/stat.h

Getting information about files.

sys/types.h

More data types.

sys/uio.h

Data types for scatter/gather calls.

sys/un.h

The address format for Unix domain sockets.

unistd.h

Prototypes for most routines defined in GUSI.

GUSI expects the Macintosh Toolbox to be initialized. This will happen automatically under some circumstances (if you're writing an `MPW` tool with the non-CodeWarrior compilers or if you are linking with `SIOW` and are forcing a write to standard output or standard error before you are using any non-file `GUSI` routines, but it's often wiser to do an explicit initialization anyway.

You should init the Toolbox in the following way:

```
InitGraf((Ptr) &qd.thePort);
InitFonts();
InitWindows();
InitMenus();
TEInit();
InitDialogs(nil);
InitCursor();
```

You have to link your program with the `GUSI` library. The exact procedure differs slightly between the `MPW C` version, the `PPCC` version, and the `CodeWarrior` version.

Linking with MPW C GUSI

For the `MPW C` version, you should link with `{CLibraries}GUSI.o`, and optionally one or several *configuration files*. Currently, the following configuration files exist:

GUSI_Everything.cfg

Include code for everything defined in `GUSI`.

GUSI_Appletalk.cfg

Include code for AppleTalk sockets.

GUSI_Internet.cfg

Include code for MacTCP sockets.

GUSI_PAP.cfg

Include code for PAP sockets.

GUSI_PPC.cfg

Include code for PPC sockets.

GUSI_Unix.cfg

Include code for Unix domain sockets.

If you don't specify any configuration files, only the file related routines will be included. It's important that these files appear *before* all other libraries.

Linking with `GUSI` doesn't free you from linking in the standard libraries, typically:

```
{Libraries}Runtime.o
{Libraries}Interface.o
{CLibraries}StdCLib.o
{Libraries}ToolLibs.o
```

Linking with PPCC GUSI

For the `PPCC` version, you should link with `{PPCLibraries}GUSI.xcoff` and if you are linking with SIOW, also with `{PPCLibraries}GUSI.xcoff`. The `PPCC` version currently doesn't support flexible configuration. Like with the `MPW C` version, `GUSI` should be first in your link, and you have to link with the standard libraries.

`GUSI` for `PPCC` makes use of Code Fragment Manager version numbers, therefore you have to specify the correct version number for `MakePEF` with the `-l` option.

```
-l "GUSI.xcoff=GUSI#0x01508000-0x01508000"
```

In case you were wondering, this encodes the version number (1.5.0) the same way as the header of a `'vers'` resource.

Linking with CodeWarrior GUSI

The easiest way to get started with a `CodeWarrior GUSI` application is by cloning from the appropriate project stationery in the Lib directory. The principle of operation is the same as with the other versions:First `GUSI.Lib`, and then the standard libraries have to be specified. To create an MPW tool with the CodeWarrior compilers, you additionally have to link with `GUSIMPW.Lib` before `GUSI.Lib`

The `CodeWarrior` version uses a new configuration mechanism that will eventually be adapted in the other versions as well:At the beginning of your application, call `GUSISetup` for the components you need. Currently, the following components are defined:

GUSISetup(GUSIwithSIOUXSockets)

Allows use of the `SIOUX` library for standard I/O.

GUSISetup(GUSIwithAppleTalkSockets)

Includes ADSP sockets.

GUSISetup(GUSIwithInternetSockets)

Includes TCP and UDP sockets.

GUSISetup(GUSIwithPAPSockets)

Includes PAP sockets.

GUSISetup(GUSIwithPPCSockets)

Includes PPC sockets.

GUSISetup(GUSIwithUnixSockets)

Includes Unix domain sockets.

If you call *`GUSIDefaultSetup()`* instead, all of the above will be included. These calls should be included right at the beginning of your *`main()`* procedure.

Warning messages, Rezzing

You will get lots of warning messages about duplicate definitions, but that's ok (Which means I can't do anything about it).

You should also rez your program with `GUSI.r`. The section *GUSI_Advanced/Resources* discusses when and how to add your own configuration resource to customize `GUSI` defaults. Don't forget that your `PowerPC` programs also need a `cfrg` resource.

Overview

This section discusses the routines common to all, or almost all communication domains. These routines return `-1` if an error occurred, and set the variable `errno` to an error code. On success, the routines return `0` or some positive value.

Here's a list of all error codes and their typical explanations. The most important of them are repeated for the individual calls.

EACCES

Permission denied:An attempt was made to access a file in a way forbidden by its file access permissions, e.g., to *`open()`* a locked file for writing.

EADDRINUSE

Address already in use:*`bind()`* was called with an address already in use by another socket.

EADDRNOTAVAIL

Can't assign requested address:*`bind()`* was called with an address which this socket can't assume, e.g., a TCP/IP address whose `in_addr` specifies a different host.

EAFNOSUPPORT

Address family not supported:You haven't linked with this socket family or have specified a nonexisting family, e.g., AF_CHAOS.

EALREADY

Operation already in progress, e.g., *`connect()`* was called twice in a row for a nonblocking socket.

EBADF

Bad file descriptor:The file descriptor you specified is not open.

EBUSY

Request for a system resource already in incompatible use, e.g., attempt to delete an open file.

ECONNREFUSED

Connection refused, e.g. you specified an unused port for a *`connect()`*

EEXIST

File exists, and you tried to open it with `O_EXCL`.

EHOSTDOWN

Remote host is down.

EHOSTUNREACH

No route to host.

EINPROGRESS

Operation now in progress. This is *not* an error, but returned from nonblocking operations, e.g., nonblocking *connect()*.

EINTR

Interrupted system call:The user pressed Command-. or *alarm()* timed out.

EINVAL

Invalid argument or various other error conditions.

EIO

Input/output error.

EISCONN

Socket is already connected.

EISDIR

Is a directory, e.g. you tried to *open()* a directory.

EMFILE

Too many open files.

EMSGSIZE

Message too long, e.g. for an UDP *send()*.

ENAMETOOLONG

File name too long.

ENETDOWN

Network is down, e.g., Appletalk is turned off in the chooser.

ENFILE

Too many open files in system.

ENOBUFS

No buffer space available.

ENOENT

No such file or directory.

ENOEXEC

Severe error with the PowerPC standard library.

ENOMEM

Cannot allocate memory.

ENOSPC

No space left on device.

ENOTCONN

Socket is not connected, e.g., neither *`connect()`* nor *`accept()`* has been called successfully for it.

ENOTDIR

Not a directory.

ENOTEMPTY

Directory not empty, e.g., attempt to delete nonempty directory.

ENXIO

Device not configured, e.g., MacTCP control panel misconfigured.

EOPNOTSUPP

Operation not supported on socket, e.g., *`sendto()`* on a stream socket.

EPFNOSUPPORT

Protocol family not supported, i.e., attempted use of ADSP on a machine that has AppleTalk but not ADSP.

EPROTONOSUPPORT

Protocol not supported, e.g., you called *`getprotobyname()`* with neither "tcp" nor "udp" specified.

ERANGE

Result too large, e.g., *`getcwd()`* called with insufficient buffer.

EROFS

Read-only file system.

ESHUTDOWN

Can't send after socket shutdown.

ESOCKTNOSUPPORT

Socket type not supported, e.g., datagram PPC toolbox sockets.

ESPIPE

Illegal seek, e.g., *`lseek()`* called for a TCP socket.

EWOULDBLOCK

Nonblocking operation would block.

EXDEV

Cross-device link, e.g. *`FSpSmartMove()`* attempted to move file to a different volume.

Creating and destroying sockets

A socket is created with *`socket()`* and destroyed with *`close()`*.

`int socket(int af, int type, int protocol)` creates an endpoint for communication and returns a descriptor. `af` specifies the communication domain to be used. Valid values are:

AF_UNIX

Communication internal to a single Mac.

AF_INET

TCP/IP, using `MacTCP`.

AF_APPLETALK

Appletalk, using ADSP.

AF_PPC

The Program-to-Program Communication Toolbox.

`type` specifies the semantics of the communication. The following two types are available:

SOCK_STREAM

A two way, reliable, connection based byte stream.

SOCK_DGRAM

Connectionless, unreliable messages of a fixed maximum length.

`protocol` would be used to specify an alternate protocol to be used with a socket. In `GUSI`, however, this parameter is always ignored.

Error codes:

EINVAL

The `af` you specified doesn't exist.

EMFILE

The descriptor table is full.

`void close(int fd)` removes the access path associated with the descriptor, and closes the file or socket if the last access path referring to it was removed.

Prompting the user for an address

To give the user the opportunity of entering an address for a socket to be bound or connected to, the *`choose()`* routine was introduced in `GUSI`. This routine has no counterpart in UNIX implementations.

C puts up a modal dialog prompting the user to choose an address. `dom` specifies the communication domain, like in `socket`. `type` may be used by future communication domains to further differentiate within a domain, but is ignored by current domains. `prompt` is a message that will appear in the dialog. `constraint` may be used to restrict the types of acceptable addresses (For more information, consult the section of the communication domain). The following two `flags` are defined for most socket types:

CHOOSE_DEFAULT

Offer the contents passed in `name` as the default choice.

CHOOSE_NEW

Prompt for a new address, suitable for passing to *`bind()`*. Default is prompting for an existing address, to be used by *`connect()`*.

`name` on input contains a default address if CHOOSE_DEFAULT is set. On output, it is set to the address chosen.

Error codes:

EINVAL

One of the `flags` is not (yet) supported by this communications domain. This error is never reported for CHOOSE_DEFAULT , which might get silently ignored.

EINTR

The user chose "Cancel" in the dialog.

Establishing connections between sockets

Before you can transmit data on a stream socket, it must be connected to a peer socket. Connection establishment is asymmetrical:The server socket registers its address with *`bind()`*, calls *`listen()`* to indicate its willingness to accept connections and accepts them by calling *`accept()`*. The client socket, after possibly having registered its address with *`bind()`* (This is not necessary for all socket families as some will automatically assign an address) calls *`connect()`* to establish a connection with a server.

It is possible, but not required, to call *`connect()`* for datagram sockets.

`int bind(int s, const struct sockaddr *name, int namelen)` binds a socket to its address. The format of the address is different for every socket family. For some families, you may ask the user for an address by calling *`choose()`*.

Error codes:

EAFNOSUPPORT

`name` specifies an illegal address family for this socket.

EADDRINUSE

There is already another socket with this address.

`int listen(int s, int qlen)` turns a socket into a listener. `qlen` determines how many sockets can concurrently wait for a connection, but is ignored for almost all socket families.

`int accept(int s, struct sockaddr *addr, int *addrlen)` accepts a connection for a socket *on a new socket* and returns the descriptor of the new socket. If `addr` is not `NULL`, the address of the connecting socket will be assigned to it.

You can find out if a connection is pending by calling *`select()`* to find out if the socket is ready for *reading*.

Error codes:

ENOTCONN

You did not call *`listen()`* for this socket.

EWOULDBLOCK

The socket is nonblocking and no socket is trying to connect.

`int connect(int s, const struct sockaddr *addr, int addrlen)` tries to connect to the socket whose address is in `addr`. If the socket is nonblocking and the connection cannot be made immediately, *`connect()`* returns EINPROGRESS . You can find out if the connection has been established by calling *`select()`* to find out if the socket is ready for *writing*.

Error codes:

EAFNOSUPPORT

`name` specifies an illegal address family for this socket.

EISCONN

The socket is already connected.

EADDRNOAVAIL

There is no socket with the given address.

ECONNREFUSED

The socket refused the connection.

EINPROGRESS

The socket is nonblocking and the connection is being established.

Transmitting data between sockets

You can write data to a socket using *write()*, *writev()*, *send()*, *sendto()*, or *sendmsg()*. You can read data from a socket using *read()*, *readv()*, *recv()*, *recvfrom()*, or *recvmsg()*.

`int read(int s, char *buffer, unsigned buflen)` reads up to `buflen` bytes from the socket. *read()* for sockets differs from *read()* for files mainly in that it may read fewer than the requested number of bytes without waiting for the rest to arrive.

Error codes:

EWOULDBLOCK

The socket is nonblocking and there is no data immediately available.

`int readv(int s, const struct iovec *iov, int count)` performs the same action, but scatters the input data into the `count` buffers of the `iov`Jarray, always filling one buffer completely before proceeding to the next. `iovec` is defined as follows:

```
struct iovec {
  caddr_t  iov_base;   /* Address of this buffer */
  int   iov_len;       /* Length of the buffer */
};
```

`int recv(int s, void *buffer, int buflen, int flags)` is identical to *read()*, except for the `flags` parameter. If the `MSG_OOB` flag is set for a stream socket that supports out-of-band data, *recv()* reads out-of-band data.

`int recvfrom(int s, void *buffer, int buflen, int flags, void *from, int *fromlen)` is the equivalent of *recv()* for unconnected datagram sockets. If `from` is not `NULL`, it will be set to the address of the sender of the message.

`int recvmsg(int s, struct msghdr *msg, int flags)` is the most general routine, combining the possibilities of *readv()* and *recvfrom()*. `msghdr` is defined as follows:

```
struct msghdr {
  caddr_t  msg_name;            /* Like from in recvfrom() */
  int      msg_namelen;         /* Like fromlen in recvfrom() */
  struct   iovec *msg_iov;      /* Scatter/gather array */
  int      msg_iovlen;          /* Number of elements in msg_iov */
  caddr_t  msg_accrights;       /* Access rights sent/received. Not
used in GUSI */
  int      msg_accrightslen;
};
```

`int write(int s, char *buffer, unsigned buflen)` writes up to `buflen` bytes to the socket. As opposed to *`read()`*, *`write()`* for nonblocking sockets always blocks until all bytes are written or an error occurs.

Error codes:

EWOULDBLOCK

The socket is nonblocking and data can't be immediately written.

`int writev(int s, const struct iovec *iov, int count)` performs the same action, but gathers the output data from the `count` buffers of the `iov`Jarray, always sending one buffer completely before proceeding to the next.

`int send(int s, void *buffer, int buflen, int flags)` is identical to *`write()`*, except for the `flags` parameter. If the `MSG_OOB` flag is set for a stream socket that supports out-of-band data, *`send()`* sends an out-of-band message.

`int sendto(int s, void *buffer, int buflen, int flags, void *to, int *tolen)` is the equivalent of *`send()`* for unconnected datagram sockets. The message will be sent to the socket whose address is given in `to`.

`int sendmsg(int s, const struct msghdr *msg, int flags)` combines the possibilities of *`writev()`* and *`sendto()`*.

I/O multiplexing

int select(int width, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout) examines the I/O descriptors specified by the bit masks `readfs`, `writefs`, and `exceptfs` to see if they are ready for reading, writing, or have an exception pending. `width` is the number of significant bits in the bit mask. *`select()`* replaces the bit masks with masks of those descriptors which are ready and returns the total number of ready descriptors. `timeout`, if not `NULL`, specifies the maximum time to wait for a descriptor to become ready. If `timeout` is `NULL`, *`select()`* waits indefinitely. To do a poll, pass a pointer to a zero `timeval` value in `timeout`. Any of `readfds`, `writefds`, or `exceptfds` may be given as `NULL` if no descriptors are of interest.

Error codes:

EBADF

One of the bit masks specified an invalid descriptor.

The descriptor bit masks can be manipulated with the following macros:

```
FD_ZERO(fds);    /* Clear all bits in *fds */
FD_SET(n, fds);  /* Set bit n in *fds */
FD_CLR(n, fds);  /* Clear bit n in *fds */
FD_ISSET(n, fds);   /* Return 1 if bit n in *fds is set, else 0 */
```

Getting and changing properties of sockets

You can obtain the address of a socket and the socket it is connected to by calling *getsockname()* and *getpeername()* respectively. You can query and manipulate other properties of a socket by calling *ioctl()*, *fcntl()*, *getsockopt()*, and *setsockopt()*. You can create additional descriptors for a socket by calling *dup()* or *dup2()*.

`int getsockname(int s, struct sockaddr *name, int *namelen)` returns in `*name` the address the socket is bound to. `*namelen` should be set to the maximum length of `name` and will be set by *getsockname()* to the actual length of the name.

`int getpeername(int s, struct sockaddr *name, int *namelen)` returns in `*name` the address of the socket that this socket is connected to. `*namelen` should be set to the maximum length of `name` and will be set by *getpeername()* to the actual length of the name.

`int ioctl(int d, unsigned int request, long *argp)` performs various operations on the socket, depending on the `request`. The following codes are valid for all socket families:

FIONBIO

Make the socket blocking if the `long` pointed to by argp is `0`, else make it nonblocking.

FIONREAD

Set `*argp` to the number of bytes waiting to be read.

Error codes:

EOPNOTSUPP

The operation you requested with `request` is not supported by this socket family.

`int fcntl(int s, unsigned int cmd, int arg)` provides additional control over a socket. The following values of `cmd` are defined for all socket families:

F_DUPFD

Return a new descriptor greater than or equal to `arg` which refers to the same socket.

F_GETFL

Return descriptor status flags.

F_SETFL

Set descriptor status flags to `arg`.

The only status flag implemented is `FNDELAY` which is true if the socket is nonblocking.

Error codes:

EOPNOTSUPP

The operation you requested with `cmd` is not supported by this socket family.

`int getsockopt(int s, int level, int optname, void *optval, int * optlen)` is used to request information about sockets. It is not implemented in `GUSI`.

`int setsockopt(int s, int level, int optname, void *optval, int optlen)` is used to set options associated with a socket. It is not implemented in `GUSI`.

`int dup(int fd)` returns a new descriptor referring to the same socket as `fd`. The old and new descriptors are indistinguishible. The new descriptor will always be the smallest free descriptor.

`int dup2(int oldfd, int newfd)` closes `newfd` if it was open and makes it a duplicate of `oldfd`. The old and new descriptors are indistinguishible.

File system calls

Files are unlike sockets in many respects:Their length is never changed by other processes, they can be rewound. There are also many calls which are specific to files.

Differences to generic behavior

The following calls make no sense for files and return an error of `EOPNOTSUPP`:

```
socket()
bind()
listen()
accept()
connect()
getsockname()
getpeername()
getsockopt()
setsockopt()
```

The following calls *will* work, but might be frowned upon by your friends (besides, UNIX systems generally wouldn't like them):

```
recv()
recvfrom()
recvmsg()
send()
sendto()
sendmsg()
```

`choose()` returns zero terminated C strings in `name`. It accepts an additional flag `CHOOSE_DIR`. If this is set, *`choose()`* will select directories instead of files.

You may restrict the files presented for choosing by passing a pointer to the following structure for the `constraint` argument:

```
typedef struct {
  short       numTypes;   /* Number of legitimate file types */
  SFTypeList  types;      /* The types, like 'TEXT' */
}sa_constr_file;
```

`select()` will give boring results. File descriptors are *always* considered ready to read or write, and *never* give exceptions.

`ioctl()` and *`fcntl()`* don't support manipulating the blocking state of a file descriptor or reading the number of bytes available for reading, but will accept lots of other requests---Check with your trusty MPW C documentation.

Routines specific to the file system

In this section, you'll meet lots of good old friends. Some of these routines also exist in the standard MPW libraries, but the `GUSI` versions have a few differences:

File names are relative to the directory specified by *chdir()*.

You can define special treatment for some file names (See below under "Adding your own file families").

You can pass `FSSpec` values to the routines by encoding them with *`FSp2Encoding()`* (See "FSSpec routines" below).

`int stat(const char * path, struct stat * buf)` returns information about a file. `struct stat` is defined as follows:

```
struct stat   {
  dev_t    st_dev;     /* Volume reference number of file  */
  ino_t    st_ino;     /* File or directory ID             */
  u_short  st_mode;    /* Type and permission of file      */
  short    st_nlink;   /* Always 1                          */
  short    st_uid;     /* Set to 0                          */
  short    st_gid;     /* Set to 0                          */
  dev_t    st_rdev;    /* Set to 0                          */
  off_t    st_size;
  time_t   st_atime;   /* Set to st_mtime    */
  time_t   st_mtime;
  time_t   st_ctime;
  long     st_blksize;
  long     st_blocks;
};
```

`st_mode` is composed of a file type and of file permissions. The file type may be one of the following:

S_IFREG

A regular file.

S_IFDIR

A directory.

S_IFLNK

A finder alias file.

S_IFCHR

A console file under MPW or SIOW.

S_IFSOCK

A file representing a UNIX domain socket.

Permissions consist of an octal digit repeated three times. The three bits in the digit have the following meaning:

4

File can be read.

2

File can be written.

1

File can be executed, i.e., its type is `APPL' or 'appe'. The definition of executability can be customized with the `GUSI_ExecHook` discussed in the advanced section.

`int lstat(const char * path, struct stat * buf)` works just like *`stat()`*, but if `path` is a symbolic link, *`lstat()`* will return information about the link and not about the file it points to.

`int fstat(int fd, struct stat * buf)` is the equivalent of *`stat()`* for descriptors representing open files. While it is legal to call *`fstat()`* for sockets, the information returned is not really interesting. The file type in `st_mode` will be S_IFSOCK for sockets.

`int chmod(const char * filename, mode_t mode)` changes the mode returned by *`stat()`*. Currently, the only thing you can do with *`chmod()`* is to turn the write permission off an on. This is translated to setting and clearing the file lock bit.

`int utime(const char * file, const struct utimbuf * tim)` changes the modification time of a file. `struct utimbuf` is defined as:

```
struct utimbuf {
  time_t actime;       /* Access time */
  time_t modtime;      /* Modification time */
};
```

`actime` is ignored, as the Macintosh doesn't store access times. The modification of `file` is set to `modtime`.

`int isatty(int fd)` returns 1 if `fd` represents a terminal (i.e. is connected to `"Dev:Stdin"` and the like), `0` otherwise.

`long lseek(int, long, int)` works the same as the `MPW` routine, and will return `ESPIPE` if called for a socket.

`int remove(const char *filename)` removes the named file. If `filename` is a symbolic link, the link will be removed and not the file.

`int unlink(const char *filename)` is identical to *`remove()`*. Note that on the Mac, *`unlink()`* on open files behaves differently from `UNIX`.

`int rename(const char *oldname, const char *newname)` renames and/or moves a file. `oldname` and `newname` must specify the same volume, but as opposed to the standard `MPW` routine, they may specify different folders.

`int open(const char*, int flags)` opens a named file. The `flags` consist of one of the following modes:

O_RDONLY

Open for reading only.

WR_ONLY

Open for writing only.

O_RDWR

Open for reading and writing.

Optionally combined with one or more of:

O_APPEND

The file pointer is set to the end of the file before each write.

O_RSRC

Open resource fork.

O_CREAT

If the file does not exist, it is created.

O_EXCL

In combination with O_CREAT , return an error if file already exists.

O_TRUNC

If the file exists, its length is truncated to 0; the mode is unchanged.

O_ALIAS

If the named file is a symbolic link, open the link, not the file it points to (This is most likely an incredibly bad idea).

`int creat(const char * name)` is identical to `open(name, O_WRONLY+O_TRUNC+O_CREAT)`. If the file didn't exist before, `GUSI` determines its file type and creator of the according to rules outlined in the section "Resources" below.

`int faccess(const char *filename, unsigned int cmd, long *arg)` works the same as the corresponding `MPW` routine, but respects calls to *`chdir()`* for partial filenames.

`void fgetfileinfo(char *filename, unsigned long *newcreator, unsigned long *newtype)` returns the file type and creator of a file.

`void fsetfileinfo(char *filename, unsigned long newcreator, unsigned long newtype)` sets the file type and creator of a file to the given values.

`int symlink(const char* linkto, const char* linkname)` creates a file named `linkname` that contains an alias resource pointing to `linkto`. The created file should be indistinguishible from an alias file created by the System 7 Finder. Note that aliases bear only superficial similiarities to `UNIX` symbolic links, especially once you start renaming files.

`int readlink(const char* path, char* buf, int bufsiz)` returns in `buf` the name of the file that `path` points to.

`int truncate(const char * path, off_t length)` causes a file to have a size equal to `length` bytes, shortening it or extending it with zero bytes as necessary.

`int ftruncate(int fd, off_t length)` does the same thing with an open file.

`int access(const char * path, int mode)` tests if you have the specified access rights to a file. `mode` may be either `F_OK`, in which case the file is tested for existence, or a combination of the following:

R_OK

Tests if file is readable.

W_OK

Tests if file is writeable.

X_OK

Tests if file is executable. As with *`stat()`*, the definition of executability may be customized.

`access()` returns 0 if the specified access rights exist, otherwise it sets `errno` and returns -1.

`int mkdir(const char * path)` creates a new directory.

`int rmdir(const char * path)` deletes an empty directory.

`int chdir(const char * path)` makes all future partial pathnames relative to this directory.

`char * getcwd(const char * buf, int size)` returns a pointer to the current directory pathname. If `buf` is `NULL`, `size` bytes will be allocated using *`malloc()`*.

Error codes:

ENAMETOOLONG

The pathname of the current directory is greater than `size`.

ENOMEM

`buf` was `NULL` and *`malloc()`* failed.

A number of calls facilitate scanning directories. Directory entries are represented by following structure:

```
    struct dirent {
      u_long   d_fileno;      /* file number of entry */
      u_short  d_reclen;      /* length of this record */
      u_short  d_namlen;      /* length of string in d_name */
    #define MAXNAMLEN   255
      char  d_name[MAXNAMLEN + 1];  /* name must be no longer than this
*/
    };
```

`DIR * opendir(const char * dirname)` opens a directory stream and returns a pointer or `NULL` if the call failed.

`struct dirent * readdir(DIR * dirp)` returns the next entry from the directory or `NULL` if all entries have been processed.

`long telldir(const DIR * dirp)` returns the position in the directory.

`void seekdir(DIR * dirp, long loc)` changes the position.

`void rewinddir(DIR * dirp)` restarts a scan at the beginning.

`int closedir(DIR * dirp)` closes the directory stream.

`int scandir(const char * path, struct dirent *** entries, int (*want)(struct dirent *), int (*sort)(const void *, const void *))` scans a whole directory at once and returns a possibly sorted list of entries. If `want` is not `NULL`, only entries for which `want` returns 1 are returned. If `sort` is not `NULL`, the list is sorted using *`qsort()`* with sort as a comparison function. If sort is `NULL`, the list will be sorted alphabetically on a Mac, but not necessarily on other machines.

Unix domain sockets

This domain is quite regular and supports all calls that work on any domain, except for out-of-band data.

Differences to generic behavior

Addresses are file system pathnames. `GUSI` complies to the Unix implementation in that it doesn't require the name to be terminated by a zero. Names that are generated by `GUSI`, however, will always be zero terminated (but the zero won't be included in the count).

```
    struct sockaddr_un {
      short    sun_family;    /* Always AF_UNIX     */
      char     sun_path[108]; /* A pathname to a file */
    };
```

```
C<choose()> works both for existing and new addresses, and no
restriction
is possible (or necessary).
```

Appletalk sockets

Currently, only stream sockets (including out-of-band data) are supported. Appletalk sockets should work between all networked Macintoshes and between applications on a single Mac, provided the SetSelfSend flag is turned on. However, PPC sockets have a better performance for interapplication communication on a single Machine.

Differences to generic behavior

Two classes of addresses are supported for AppleTalk. The main address type specifies numeric addresses.

```
    struct sockaddr_atlk {
      short       family;     /* Always AF_APPLETALK                 */
      AddrBlock   addr;       /* The numeric AppleTalk socket address
*/
    };
```

For *`bind()`* and *`connect()`*, however, you are also allowed to specify symbolic addresses. *`bind()`* registers an NBP address, and *`connect()`* performs an NBP lookup. Registered NBP adresses are automatically released when the socket is closed. No call ever *returns* a symbolic address.

```
    struct sockaddr_atlk_sym {
      short       family;     /* Always ATALK_SYMADDR     */
      EntityName  name;       /* The symbolic NBP address   */
    };
```

`choose()` currently only works for existing sockets. The peer must have registered a symbolic address. To restrict the choice of addresses presented, pass a pointer to the following structure for the `constraint` argument:

```
    typedef struct {
      short       numTypes;   /* Number of allowed types */
      NLType      types;      /* List of types */
    }sa_constr_atlk;
```

PPC sockets

These provide authenticated stream sockets without out-of-band data. PPC sockets should work between all networked Macintoshes running System 7, and between applications on a single Macintosh running System 7.

Differences to generic behavior

PPC socket addresses have the following format:

```
    struct sockaddr_ppc {
      short             family;     /* Always AF_PPC
                 */
      LocationNameRec   location;   /* Check your trusty Inside
Macintosh  */
      PPCPortRec        port;
    };
```

choose() currently only works for existing sockets. To restrict the choice of addresses presented, pass a pointer to the following structure for the `constraint` argument:

```
    typedef struct   {
      short       flags;
      Str32       nbpType;
      PPCPortRec  match;
    }sa_constr_ppc;
```

`flags` is obtained by or'ing one or several of the following constants:

PPC_CON_NEWSTYLE

Always required for compatibility reasons.

PPC_CON_MATCH_NBP

Only display machines that have registered an entity of type `nbpType`.

PPC_CON_MATCH_NAME

Only display ports whose name matches `match.name`.

PPC_CON_MATCH_TYPE

Only display ports whose type matches `match.u.portType`.

`nbpType` specifies the machines to be displayed, as explained above. `match` contains the name and/or type to match against.

connect() will block even if the socket is nonblocking. In practice, however, delays are likely to be quite short, as it never has to block on a higher level protocol and the PPC ToolBox will automatically establish the connection.

Internet sockets

These are the real thing for real programmers. Out-of-band data only works for sending. Both stream (TCP) and datagram (UDP) sockets are supported. Internet sockets are also suited for interapplication communication on a single machine, provided it runs MacTCP.

Differences to generic behavior

Internet socket addresses have the following format:

```
    struct in_addr {
      u_long s_addr;
    };
    struct sockaddr_in {
       u_char   sin_len;          /* Ignored */
       u_char   sin_family;       /* Always C<AF_INET> */
       u_short  sin_port;         /* Port number */
       struct   in_addr sin_addr; /* Host ID */
       char     sin_zero[8];
    };
```

Routines specific to TCP/IP sockets

There are several routines to convert between numeric and symbolic addresses.

Hosts are represented by the following structure:

```
    struct hostent {
      char *h_name;        /* Official name of the host              */
      char **h_aliases;    /* A zero terminated array of alternate names
for the host     */
      int  h_addrtype;     /* Always AF_INET                        */
      int  h_length;       /* The length, in bytes, of the address   */
      char **h_addr_list;  /* A zero terminated array of network
addresses for the host   */
    };
```

`struct hostent * gethostbyname(char *name)` returns an entry for the host with the given `name` or `NULL` if a host with this name can't be found.

`struct hostent * gethostbyaddr(const char *addrP, int, int)` returns an entry for the host with the given address or `NULL` if a host with this name can't be found. `addrP` in fact has to be a `struct in_addr *`. The last two parameters are ignored.

`char * inet_ntoa(struct in_addr inaddr)` converts an internet address into the usual numeric string representation (e.g., 0x8184023C is converted to "129.132.2.60")

`struct in_addr inet_addr(char *address)` converts a numeric string into an internet address (If `x` is a valid address, `inet_addr(inet_ntoa(x)) == x`).

`int gethostname(char *machname, long buflen)` gets our name into `buffer`.

Services are represented by the following data structure:

```
struct  servent {
  char  *s_name;       /* official service name */
  char  **s_aliases;   /* alias list  */
  int   s_port;        /* port number */
  char  *s_proto;      /* protocol to use ("tcp" or "udp") */
};
```

`void setservent(int stayopen)` rewinds the file of services. If `stayopen` is set, the file will remain open until *`endservent()`* is called, else it will be closed after the next call to *`getservbyname()`* or *`getservbyport()`*.

`void` *`endservent()`* closes the file of services.

`struct servent *` *`getservent()`* returns the next service from the file of services, opening the file first if necessary. If the file is not found (`/etc/services` in the preferences folder), a small built-in list is consulted. If there are no more services, *`getservent()`* returns `NULL`.

`struct servent * getservbyname (const char * name, const char * proto)` finds a named service by calling *`getservent()`* until the protocol matches `proto` and either the name or one of the aliases matches `name`.

`struct servent * getservbyport (int port, const char * proto)` finds a service by calling *`getservent()`* until the protocol matches `proto` and the port matches `port`.

Protocols are represented by the following data structure:

```
struct  protoent {
  char  *p_name;       /* official protocol name */
  char  **p_aliases;   /* alias list (always NULL for GUSI)*/
  int   p_proto;       /* protocol number */
};
```

`struct protoent * getprotobyname(char * name)` finds a named protocol. This call is rather unexciting.

PAP sockets

PAP, the AppleTalk Printer Access Protocol is a protocol which is almost exclusively used to access networked printers. The current implementation of PAP in `GUSI` is quite narrow in that it only implements the workstation side of PAP and only in communication to the currently selected LaserWriter. It is also doomed, as it depends on Apple system resources that probably are not supported anymore in Apple's Quickdraw GX printing architecture, but if there is enough interest, the current implementation might be replaced some time.

Routines specific to PAP sockets

While PAP sockets behave in most respects like other sockets, they can currently not be created with the *`socket()`* call, but are opened with *`open()`*.

`int open("Dev:Printer", int flags)` opens a connection to the last selected LaserWriter. `flags` is currently ignored.

Communication with LaserWriters is somewhat strange. The three main uses of PAP sockets are probably interactive sessions, queries, and downloads, which will be discussed in the following sections. As in all other socket families, `GUSI` does no filtering of the transmitted data, which means that lines sent by the LaserWriter will be separated by linefeeds (ASCII 10) rather than carriage returns (ASCII 13), which are used for this purpose in most other Mac contexts. For data you *send*, it doesn't matter which one you use.

You start an *interactive session* by sending a line `"executive"` after opening the socket. This will put lots of LaserWriters (certainly all manufactured by Apple, but probably not a Linotronic) into interactive mode. If you want to, you can now play terminal emulator and use your LaserWriter as an expensive desk calculator.

A *query* is some PostScript code you send to a LaserWriter that you expect to be answered. This is quite straightforward, except that LaserWriters don't seem to answer until you have indicated to them that no more data from you will be coming. Therefore, you have to call `shutdown(s,1)` to shut the socket down for writing after you have written your query and before you try to read the answer. The following code demonstrates how to send a query to the printer:

```
int s = open("Dev:Printer", O_RDWR);
write(s, "FontDirectory /Gorilla-SemiBold exch known...", len);
/* We won't write any more */
shutdown(s, 1);
while(read(s, buf, len) > 0)
   do_something();
close(s);
```

If you want to simply *download* a file, you can also ignore the LaserWriter's response and simply close the socket after downloading.

Miscellaneous

BSD memory routines

These are implemented as macros if you

```
#include <compat.h>
```

`void bzero(void * from, int len)` zeroes `len` bytes, starting at `from`.

`void bfill(void * from, int len, int x)` fills `len` bytes, starting at `from` with `x`.

`void bcopy(void * from, void * to, int len)` copies `len` bytes from `from` to `to`.

`int bcmp(void * s1, void * s2, int len)` compares `len` bytes at `s1` against `len` bytes at `s2`, returning zero if the two areas are equal, nonzero otherwise.

Hooks

You can override some of GUSI's behaviour by providing hooks to GUSI. Note that these often get called from deep within GUSI, so be sure you understand what is required of a hook before overriding it.

GUSI hooks can be accessed with the following routines:

```
typedef void (*GUSIHook)(void);
void GUSISetHook(GUSIHookCode code, GUSIHook hook);
GUSIHook GUSIGetHook(GUSIHookCode code);
```

Currently, two hooks are defined. The `GUSI_SpinHook` is defined in the next section. The `GUSI_ExecHook` is used by GUSI to decide whether a file or folder is to be considered "executable" or not. The default hook considers all folders and all applications (i.e., files of type `'APPL'` and `'appe'` to be executable. To provide your own hook, call

```
GUSISetHook(GUSI_ExecHook, (GUSIHook) my_exec_hook);
```

where `my_exec_hook` is defined as

```
Boolean my_exec_hook(const GUSIFileRef & ref);
```

The old value is available as:

```
Boolean (*)(const GUSIFileRef & ref)GUSIgetHook(GUSI_ExecHook);
```

Blocking calls

Since the Macintosh doesn't have preemptive task switching, it is important that other applications get a chance to run during blocking calls. This section discusses the mechanism `GUSI` uses for that purpose.

While a routine is waiting for a blocking call to terminate, it repeatedly calls a spin routine with the following parameters:

```
   typedef enum spin_msg
   {
      SP_MISC,          /* some weird thing, usually just return
immediately if you get this */
      SP_SELECT,        /* in a select call, passes ticks the program is
prepared to wait */
      SP_NAME,          /* getting a host by name */
      SP_ADDR,          /* getting a host by address */
      SP_STREAM_READ,   /* Stream read call */
      SP_STREAM_WRITE,  /* Stream write call */
      SP_DGRAM_READ,    /* Datagram read call */
      SP_DGRAM_WRITE,   /* Datagram write call */
      SP_SLEEP,         /* sleeping, passes ticks left to sleep */
      SP_AUTO_SPIN      /* Automatically spinning, passes spin count */
   }spin_msg;
   typedef int (*GUSISpinFn)(spin_msg msg, long param);
```

If the spin routine returns a nonzero value, the call is interrupted and `EINTR` returned. You can modify the spin routine with the following calls:

```
GUSISetHook(GUSI_SpinHook, (GUSIHook) my_spin_hook);
(GUSISpinFn)GUSIGetHook(GUSI_SpinHook);
```

(For backward compatibility, GUSI also defines the equivalents:)

```
int GUSISetSpin(GUSISpinFn routine);
GUSISpinFn GUSIGetSpin(void);
```

Often, however, the default spin routine will do what you want:It spins a cursor and occasionally calls *`GetNextEvent()`* or *`WaitNextEvent()`*. By default, only mouse down and suspend/resume events are handled, but you can change that by passing your own `GUSIEvtTable` to *`GUSISetEvents()`*.

```
int GUSISetEvents(GUSIEvtTable table);
GUSIEvtHandler * GUSIGetEvents(void);
```

A GUSIEvtTable is a table of GUSIEvtHandlers, indexed by event code. Presence of a non-nil entry in the table will cause that event class to be allowed for *GetNextEvent()* or *WaitNextEvent()*. GUSI for MPW C and PPCC includes one event table to be used with the SIOW library.

```
typedef void (*GUSIEvtHandler)(EventRecord * ev);
typedef GUSIEvtHandler GUSIEvtTable[24];
extern GUSIEvtHandler   GUSISIOWEvents[];
```

GUSI also supports three POSIX/BSD routines:`alarm(unsigned sec)` will after sec seconds cancel the current call, raise SIGALRM, and return EINTR. Note that the default handler for SIGALRM terminates the program, so be sure to install your own handler. `alarm(0)` cancels an alarm and returns the remaining seconds. As opposed to POSIX systems, the GUSI version of *alarm()* does not use real clock interrupts and merely interrupts during a blocking call.

`sleep(unsigned sec)` sleeps for sec seconds, and `usleep(unsigned usec)` does the same for usec micorseconds (rounded to 60ths of a tick).

Resources

A few GUSI routines (currently primarily *choose()*) need resources to work correctly. These are added if you Rez your program with GUSI.r. On startup, GUSI also looks for a *preference* resource with type 'GUZI' (the 'Z' actually must be a capital Sigma) and ID GUSIRsrcID, which is currently defined as follows:

```
#ifndef GUSI_PREF_VERSION
#define GUSI_PREF_VERSION '0102'
#endif
type 'GUZI' {
   literal longint   text  =  'TEXT';  /* Type for creat'ed files
      */
   literal longint   mpw   =  'MPS ';  /* Creator for creat'ed files
   */
   byte     noAutoSpin, autoSpin;      /* Automatically spin cursor ?
  */
#if GUSI_PREF_VERSION >= '0110'
   boolean  useChdir, dontUseChdir;    /* Use chdir() ?
            */
   boolean  approxStat, accurateStat;  /* statbuf.st_nlink = # of
subdirectories ? */
   boolean  noTCPDaemon, isTCPDaemon;  /* Inetd client ?
            */
   boolean  noUDPDaemon, isUDPDaemon;
#if GUSI_PREF_VERSION >= '0150'
   boolean  noConsole, hasConsole;     /* Are we providing our own
dev:console ? */
   fill     bit[3];
#else
```

```
   fill     bit[4];
#endif
   literal longint = GUSI_PREF_VERSION;
#if GUSI_PREF_VERSION >= '0120'
   integer = @t$$@>Countof(SuffixArray);
   wide array SuffixArray {
         literal longint;              /* Suffix of file */
         literal longint;              /* Type for file */
         literal longint;              /* Creator for file */
   };
#endif
#endif
};
```

To keep backwards compatible, the preference version is included, and you are free to use whatever version of the preferences you want by defining `GUSI_PREF_VERSION`.

The first two fields define the file type and creator, respectively, to be used for files created by `GUSI`. The type and creator of existing files will never be changed unless explicitely requested with *fsetfileinfo()*. The default is to create text files (type `TEXT') owned by the `MPW Shell` (creator `MPS '). If you request a preference version of 1.2.0 and higher, you are also allowed to specify a list of suffixes that are given different types. An example of such a list would be:

```
{'SYM ', 'MPSY', 'sade'  }
```

The `autoSpin` value, if nonzero, makes `GUSI` call the spin routine for every call to *`read()`*, *`write()`*, *`send()`*, or *`recv()`*. This is useful for making an I/O bound program MultiFinder friendly without having to insert explicit calls to *`SpinCursor()`*. If you don't specify a preference resource, `autoSpin` is assumed to be `1`. You may specify arbitrary values greater than one to make your program even friendlier; note, however, that this will hurt performance.

The `useChdir` flag tells `GUSI` whether you change directories with the toolbox calls *`PBSetVol()`* or *`PBHSetVol()`* or with the `GUSI` call *`chdir()`*. The current directory will start with the directory your application resides in or the current `MPW` directory, if you're running an `MPW` tool. If `useChdir` is specified, the current directory will only change with *`chdir()`* calls. If `dontUseChdir` is specified, the current directory will change with toolbox calls, until you call *`chdir()`* the first time. This behaviour is more consistent with the standard `MPW` library, but has IMHO no other redeeming value. If you don't specify a preference resource, `useChdir` is assumed.

If `approxStat` is specified, *`stat()`* and *`lstat()`* for directories return in `st_nlink` the number of *items* in the directory `+ 2`. If `accurateStat` is specified, they return the number of *subdirectories* in the directory. The latter has probably the best chances of being compatible with some Unix software, but the former is often a sufficient upper bound, is much faster, and most programs don't care about this value anyway. If you don't specify a preference resource, `approxStat` is assumed.

The `isTCPDaemon` and `isUDPDaemon` flags turn `GUSI` programs into clients for David Petersons `inetd`, as discussed below. If you don't specify a preference resource, `noTCPDaemon` and `noUDPDaemon` are assumed.

The `hasConsole` flag should be set if you are overriding the default "dev:console", as discussed below.

Advanced techniques

This section discusses a few techniques that probably not every user of GUSI needs.

FSSpec routines

If you need to do complicated things with the Mac file system, the normal GUSI routines are probably not sufficient, but you still might want to use the internal mechanism GUSI uses. This mechanism is provided in the header file TFileSpec.h, which defines both C and C++ interfaces. In the following, the C++ member functions will be discussed and C equivalents will be mentioned where available.

OSErr TFileSpec::*Error()* returns the last error provoked by a TFileSpec member function.

TFileSpec::TFileSpec(const FSSpec & spec, Boolean useAlias = false) constructs a TFileSpec from an FSSpec and resolves alias files unless useAlias is true. (The useAlias parameter is also present in the following routines, but will not be shown anymore).

TFileSpec(short vRefNum, long parID, ConstStr31Param name) constructs a TFileSpec from its components.

TFileSpec(short wd, ConstStr31Param name) constructs a TFileSpec from a working directory reference number and a path component.

This routine is available to C users as OSErr WD2FSSpec(short wd, ConstStr31Param name, FSSpec * desc).

TFileSpec(const char * path) constructs a TFileSpec from a full or relative path name. This routine is available to C users as OSErr Path2FSSpec(const char * path, FSSpec * desc).

TFileSpec(OSType object, short vol = kOnSystemDisk, long dir = 0) constructs special TFileSpecs, depending on object.

This routine is available to C users as OSErr Special2FSSpec(OSType object, short vol, long dirID, FSSpec * desc).

All constants in Folders.h acceptable for *FindFolder()* can be passed, e.g. the following:

kSystemFolderType

The system folder.

kDesktopFolderType

The desktop folder; objects in this folder show on the desk top.

kExtensionFolderType

Finder extensions go here.

kPreferencesFolderType

Preferences for applications go here.

Furthermore, the value `kTempFileType` is defined, which creates a temporary file in the temporary folder, or, if `dir` is nonzero, in the directory you specify.

`TFileSpec(short fRefNum)` constructs a `TFileSpec` from the file reference number of an open file.

In principle, a `TFileSpec` should be compatible with an `FSSpec`. However, to be absolutely sure, you can call `TFileSpec::Bless()` which will call `FSMakeFSSpec()` before passing the TFileSpec to a `FSp` file system routine.

`char * TFileSpec::FullPath()` returns the full path name of the file. The address returned points to a static buffer, so it will be overwritten on further calls. This routine is available to `C` users as `char * FSp2FullPath(const FSSpec * desc)`.

`char * TFileSpec::RelPath()` works like `FullPath()`, but when the current folder given by `chdir()` is a pparent folder of the object, a relative path name will be returned. The address returned points to a static buffer, so it will be overwritten on further calls. This routine is available to `C` users as `char * FSp2RelPath(const FSSpec * desc)`.

`char * TFileSpec::Encode()` returns an ASCII encoding which may be passed to all `GUSI` routines taking path names. The address returned points to a static buffer, so it will be overwritten on further calls. This generates short names which may be parsed rather quickly. This routine is available to `C` users as `char * FSp2Encoding(const FSSpec * desc)`.

`OSErr TFileSpec::CatInfo(CInfoPBRec & info, Boolean dirInfo = false)` Gives information about the current object. If `dirInfo` is `true`, gives information about the current object's directory. This routine is available to `C` users as `OSErr FSpCatInfo(const FSSpec * desc, CInfoPBRec * info)`.

`OSErr TFileSpec::Resolve(Boolean gently = true)` resolve the object if it is an alias file. If gently is `true` (the default), nonexisting files are tolerated.

Boolean TFileSpec::*Exists()* returns `true` if the object exists.

Boolean TFileSpec::IsParentOf(const TFileSpec & other) returns `true` if the object is a parent of `other`.

`TFileSpec TFileSpec::operator--()` replaces the object with its parent directory. This routine is available to `C` users as `OSErr FSpUp(FSSpec * desc)`.

`TFileSpec FileSpec::operator-=(int levels)` is equivalent to calling `-- levels` times and `TFileSpec FileSpec::operator-(int levels)` is equivalent to calling `-=` on a *copy* of the current object.

`TFileSpec TFileSpec::operator+=(ConstStr31Param name)`, `TFileSpec TFileSpec::operator+=(const char * name)`, and their non-destructive counterparts `+` add a further component to the current object, which must be an existing directory.

This routine is available to `C` users as `OSErr FSpDown(FSSpec * desc, ConstStr31Param name)`.

`TFileSpec TFileSpec::operator[](short index)` returns the `index`th object in the parent folder of the current object.

A destructive version of this routine is available to `C` users as `OSErr FSpIndex(FSSpec * desc, short index)`.

Furthermore, the `==` and `!=` operators are defined to test `TFileSpecs` for equality.

`OSErr FSpSmartMove(const FSSpec * from, const FSSpec * to)` does all the work of moving and renaming a file (within the same volume), handling (I hope) all special cases (You might be surprised how many there are).

File pattern iterators

Sometimes you might find it useful to find all files ending in `.h` or all directories starting with `MW`. For this purpose, `GUSI` offers a mechanism in the header file `TFileGlob.h`, which defines both `C` and `C++` interfaces.

You start a search by constructing a file pattern iterator with `TFileGlob::TFileGlob(const char * pattern, const TFileSpec * startDir = nil)`. `pattern` is an absolute or relative path name, with the following characters getting a special interpretation:

?

Matches an arbitrary single character.

*****

Matches any number of characters (including none).

\

Suppresses the special interpretation of the following character.

`startDir` provides a nonstandard starting directory for relative patterns. After you have constructed the iterator, you can check whether a file was found by calling Boolean TFileGlob::*Valid()* . If one was found, you can use the . To get the next file, call Boolean TFileGlob::*Next()* , which again returns `true` if another match was found.

To call the file pattern iterator routines from `C`, you have the following routines:

FileGlobRef NewFileGlob(const char * pattern)

Constructs an iterator.

Boolean NextFileGlob(FileGlobRef glob)

Advances the iterator.

Boolean FileGlob2FSSpec(FileGlobRef glob, FSSpec * spec)

Copies the file specification to `spec` and returns whether the iterator is valid.

void DisposeFileGlob(FileGlobRef glob)

Destructs the iterator.

Adding your own socket families

It is rather easy to add your own socket types to `GUSI`:

Pick an unused number between 17 and `GUSI_MAX_DOMAINS` to use for your address family.

Include `GUSI_P.h`.

Write a subclass of `SocketDomain` and override *`socket()`* and optionally *`choose()`*.

Write a subclass of `Socket` and override whatever you want. If you override *`recvfrom()`* and *`sendto()`*, *`read()`* and *`write()`* are automatically defined.

For more information, study the code in `GUSIDispatch.cp` and `GUSISocket.cp`, which implement the generic socket code. The easiest actual socket implementation to study is probably `GUSIUnix.cp`.

Adding your own file families

GUSI also supports adding special treatment for certain file names to almost all (tell me if I have forgotten one) standard C library routines dealing with file names. To avoid countless rescanning of file names, GUSI preprocesses the names:

If the file name starts with "Dev:" (case insensitive), the file name is considered a *device name*, and the rest of the name can have any structure you like.

Otherwise, the name is translated into a FSSpec, and therefore should refer to a real file system object (all intermediate path name components should refer to existing directories).

To create a file family:

Pick an address family, as described above. However, if you don't plan on creating sockets of this family with *socket()*, just specify AF_UNSPEC.

Include GUSIFile_P.h.

Write a subclass of FileSocketDomain, specifying whether you are interested in device names, file names, or both, and override *Yours()* and other calls.

Write a subclass of Socket and override whatever you want.

For more information, study the code in GUSIFile.cp, which implements the generic file socket code.

In your *Yours()* member function, you specify whether you are prepared to handle one of the following functions for a given file name:

```
enum Request {
  willOpen,
  willRemove,
  willRename,
  willGetFileInfo,
  willSetFileInfo,
  willFAccess,
  willStat,
  willChmod,
  willUTime,
            willAccess
};
```

If you return `true` for a request, your corresponding member function will be called. Member functions are similar to the corresponding `C` library functions, except that their first parameter is a `GUSIFileRef &` instead of a `const char *` (but further file name parameters, as in *`rename()`*, will be left untouched). You might also return `true` but *not* override the member function to indicate that standard file treatment (`EINVAL` for many routines) is OK.

The member function will always be called immediately after the *`Yours()`* function, so you may want to pre-parse the file name in the *`Yours()`* function and keep the information for the member function.